Concepts for Efficient Flow Solvers based on Adaptive Cartesian Grids

Ioan Lucian Muntean, Miriam Mehl, Tobias Neckel, and Tobias Weinzierl

Abstract This contribution describes mathematical and algorithmic concepts that allow for a both numerically and hardware efficient implementation of a flow solver. In view of numerical efficiency, this strongly suggests multigrid solvers on adaptively refined grids in order to minimise the amount of data to be computed for a prescribed accuracy as well as the number of iterations. In view of hardware efficiency, a minimisation of memory requirements and an optimisation of data structures and data access tailored to the memory hierarchy of supercomputing architectures is essential, since flow solvers typically are data intensive applications. We address both the numerical and the hardware challenge with a combination of structured but flexible adaptive hierarchical Cartesian grids with space-filling curves as traversal scheme and stacks as data structures. These basic concepts are applied to the two computationally demanding application areas turbulent flow simulations and fluid-structure interactions. We show the benefits of our methods for these applications as well as first results achieved at the HLRB2 and smaller clusters.

1 Introduction

Our task is to port state-of-the-art numerical algorithms, i.e. modern numerical schemes on adaptive grids realised with low memory requirements and with high performance, to supercomputers in order to meet the high data and performance requirements of challenging application areas such as turbulence simulation or multiphysics problems. This comprises more than using hardware-optimised software components (`METIS`, `BLAS`, `LINPACK`, ...) in a suitable way as design pattern at a higher abstraction level are required.

Ioan Lucian Muntean, Miriam Mehl, Tobias Neckel, and Tobias Weinzierl
Department of Computer Science, TU München, Boltzmannstr. 3, 85748 Garching, Germany
e-mail: `{muntean,mehl,neckel,weinzier}@in.tum.de`

First, the generation of time-dependent adaptive grids is complex and requires dynamic data structures tailored to the concrete application and algorithm. Second, such grids cannot be traversed with an i, j, k-like indexing anymore. Instead, a more sophisticated but, nevertheless, efficient scheme is required. Third, solvers for partial differential equations typically deal with sparse systems of equations. The particular sparsity pattern should be exploited. In many cases, the explicit assembly of the matrix is unnecessary and too memory consuming. Fourth, multilevel methods induce multiple connections between different grid levels. Such, also the interplay between the levels has to be treated in a sophisticated and hardware-optimal way.

It becomes obvious already from this incomplete listing of requirements, that, in general[1], there is a downward drift of the performance measured in MFLOPS or parallel efficiency, if we implement more and more sophisticated mathematical methods in our PDE solvers. However, we should implement them on supercomputers for at least two reasons: First, it is more profitable to apply for example a multigrid solver with a constant and low iteration number of let's say five and a parallel efficiency of only 80% than a conjugate gradient solver with a parallel efficiency of 99% yet with an increasing (i.e. hundreds or thousands) number of iterations for a finer grid. To achieve a reasonable hardware efficiency, overall concepts for the whole algorithm have to be established.

If we translate the requirements mentioned above into more concrete challenges for a numerically and hardware efficient PDE solver, we can identify at least five of them: First, the algorithm and, in particular, the data access mechanisms have to be fast on the actual hardware. Second, the solver has to scale on a parallel architecture[2]. Third, it has to come along with low memory requirements as, even on a shared memory machine such as the HLRB2, working solely on the local physical memory reduces runtime considerably [26, 12] and, in addition, with all the multithreaded computing cores coming up, memory access, i.e. the memory bandwidth and the cache size, becomes a bottleneck for data intensive applications such as PDE solvers. Fourth, the solver has to handle complex and – even more challenging – changing grids. Fifth, the underlying data structures and algorithmic principles have to be well-suited for multi-scale and multigrid algorithms.

In the recent years, many approaches towards hardware and storage optimisation of PDE solvers have been made: Strategies enhancing the instruction-level parallelism (such as loop unrolling), data layout optimisations (array padding, array merging) and data access optimisations (loop fusion and/or loop blocking) as well as enhanced methods implementing cache-aware versions of efficient new numerical schemes such as patch-adaptive relaxation [7].

In contrast to optimising storage access for fixed grids, we present a method that works without also for dynamically adaptive grids without loosing efficiency in dependence on the adaptivity pattern of the grid and with minimal storage require-

[1] There are exceptions such as p-methods that are inherently cache-efficient due to the usage of dense matrices, but the general trend is towards more complex grid structures and data access mechanisms and, thus, a worse memory performance.

[2] We restrict ourselves to clusters here but keep the developments in the multicore/hyperthreading community in mind during the development of our basic concepts.

ments. Our concept covers all aspects of the solver starting from data structures over data access to parallelisation and load balancing. It only depends on general properties of the hardware, such as the existence of caches or a software pipeline etc., but not on the exact hardware parameters such as cache sizes or bandwidths. In this sense, it is hardware-oblivious instead of hardware-aware.

In the following, we describe our grids, their parallel generation and traversal process (Sect. 2) and present the basic solver algorithm including data structures, multigrid solvers, dynamic adaptivity, parallelisation, extensions for a flow solver (Sect. 3), and our reference solver F3F working on a regular Cartesian grid (Sect. 4). In Sects. 5 – 7, we show results ranging from parallel flow simulations on the HLRB2 over flow simulations on adaptively refined grids performed on smaller computers to a fluid-structure interaction application example. We close the discussion with a short conclusion (Sect. 8).

2 Grids and Grid Generation

2.1 Spacepartitioning Grids

The fundamental difficulty of our work is to combine flexible and, even more severe, dynamic adaptivity of the computational grid with hardware and, in particular, memory efficiency. We overcome this difficulty using spacepartitioning trees as a construction principle of our computational, spacepartitioning grids.

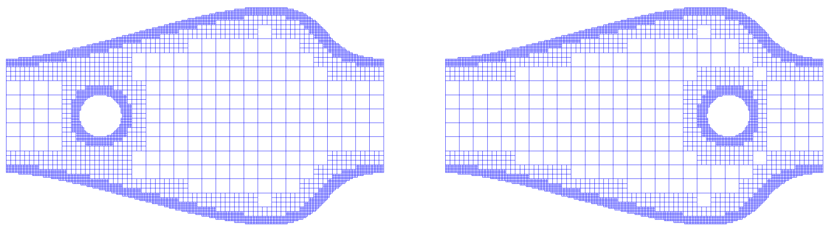

Fig. 1 Two-dimensional adaptive grids used for the simulation of particle transport in a pore with oscillating diameter. The grids describe the computational domain at different times using an Eulerian approach in combination with dynamically adaptive grid refinement.

The starting point of the construction of our grids – the root of the spacepartitioning tree – is a hypercube $[0,1]^d$ $,d \in \{2,3\}$. Our computational domain is embedded into this hypercube. In the following steps, the grid is refined in a recursive way, splitting up each cell into three parts along every coordinate axis[3]. The depth of recursion and, hence, the resolution depend on the boundary approximation and the numerical accuracy to be obtained. Complicated and changing geometries are repre-

[3] The reasons for the partitioning into three instead of two parts per coordinate direction will be explained in Sect. 3.

sented with the help of cell-markers that can change with the geometry's movement whereas the grid itself remains the same up to dynamic adaptivity (Eulerian approach [13, 22], Fig. 1). The underlying tree structure yields a high structuredness of the grid, that is a fixed and known relation between the grid elements such as vertices, edges, faces, and cells, but, at the same time, leads to a flexible and local grid adaptivity. It facilitates a domain decomposition approach for parallelisation (Sect. 2.3) and is well-suited for dynamic refinement: In each time step, we track given adaptivity criteria [17] and changes of the geometry and accordingly update the computational grid, i.e. we coarsen the grid wherever possible and add new subtrees wherever necessary. In addition, the levels of the tree can be interpreted as a multi-scale representation of the computational domain. Multigrid algorithms benefit from this fact. Concerning memory efficiency, such grids make the storage of geometric information such as coordinates or the structure's connections obsolete. Furthermore, if we use a suitable solver algorithm, the storage of global system matrices or of specialised difference stencils at boundaries between different local refinement depths is not necessary. This leads to very low overall memory requirements as shown in Table 1.

	$\frac{bytes}{cell}$	$\frac{bytes}{vertex}$	Explanation
$d = 2$	6	2	grid without degrees of freedom
$d = 2$	14	20	release version flow solver
$d = 3$	10	2	grid without degrees of freedom
$d = 3$	18	28	release version flow solver

Table 1 Memory requirements in partially staggered spacepartitioning grids as used in the flow solver (see Sect. 3.4).

2.2 Grid Generation from a Surface Representation of the Geometry

To create the grid, a discrete or continuous geometry representation has to be mapped onto the spacepartitioning data structure. As most CAD applications' surface representations support triangulations, we use a triangle mesh as original geometry description, too. This mesh is mapped onto the Peano spacetree via the marker-and-cell approach [13, 22].

For this purpose, we have to check if cells intersect with surface triangles during the grid construction to decide whether to refine the grid and how to set the cell-markers. These intersection tests can be done very efficently exploiting the recursive structure and locality awareness of the spacepartitioning tree ([18, 6]): Every tree node's geometry element is completely contained in the geometric volume represented by its father. Hence, if a triangle does not intersect the father's geometric element, it does not intersect any son's element. So, the intersection test can be done top–down in the tree using inheritance of cell information.[4] We end up with a very fast grid generation process (Table 2).

# grid vertices	$52,662,337$	$210,666,753$	$842,687,105$
runtime (sec)	48.188	168.641	662.797

Table 2 Runtimes for the generation of adaptive grids for a sphere geometry. The computations were performed on a Pentium 4 2.4GHz processor with 1GB RAM [6].

2.3 Load Balancing and Parallel Grid Generation and Traversal

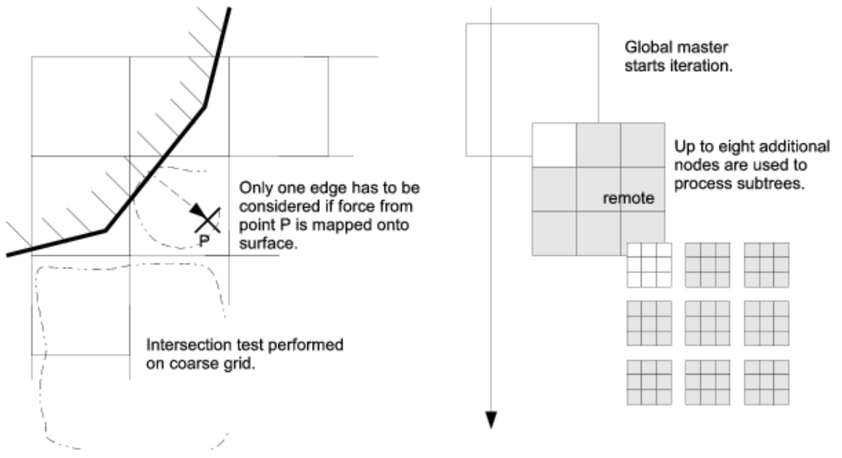
Only one edge has to be considered if force from point P is mapped onto surface.
P
Intersection test performed on coarse grid.
Global master starts iteration.
Up to eight additional nodes are used to process subtrees.
remote

Fig. 2 Efficient bidirectional mapping of triangulated surface onto the spacepartitioning grid (left) and multiscale domain decomposition approach for the parallel code (right).

It is essential for a parallel code that there is no sequential part or the need to hold data of the whole scenario on a single computing node. For PDE solvers, this implies a parallel grid generation. We exploit the fact that every tree level yields a domain decomposition. Thus, we can perform a recursive top-down splitting that fits to the depth-first traversal during the grid generation: The tree's root node is processed on a single cluster node – the global master controlling the whole CFD application. The second tree level consists of 3^d elements. Every element again is the root node of a subtree. Depending on the size of this subtree, the second level's elements are distributed among the cluster's processors, i.e. at most $3^d - 1$ additional cluster nodes are used on this level throughout the depth-first iteration. On the subsequent levels, this domain splitting procedure is applied recursively (see Fig. 2). The size of the subtrees is determined during the depth-first traversal's bottom-up steps as the

[4] For the nearest neighbour search required for the force and surface translation projection in fluid-structure interaction problems, a similar approach leads to a bottom-up algorithm.

size of a tree is an analysed tree attribute [15]. The value is stored within the tree's nodes where necessary.

If the grid is modified during the simulation, the subtrees' sizes change. This might require a merging of subtrees stored on different cluster nodes or a further splitting of subtrees to ensure a good load balancing. In [14], we examined a global load balancing. The evaluation of alternative approaches and the integration into the fluid solver's source code are work in progress.

To ensure a good subdomain layout the order of the grid cells is decisive: If the element's subcells are consecutively distributed among cluster nodes, the order defines the layout of the subpartitions. Hereby, two tasks have to be tracked [1]: First, the partition assigned to one cluster node should be connected. Second, the relation between the partition's surface and the partition's volume should be small. This minimises the amount of data to be interchanged with other nodes per work unit of the respective process. Our spacetree is based upon a 3^d subdivision and the child order is defined by the Peano space-filling curve. The Peano curve is a self-similar recursively defined space-filling curve [21]. Such curves fit to the construction process of our grids very well: The starting point is a given template prescribing the order of the partitions of a very coarse decomposition of the hypercube (see the left picture in Fig. 4 for the template of the Peano curve). In the next steps, each subdomain is decomposed again applying the template – mirrored or rotated – again (see the middle picture in Fig. 4 for the first iteration of the Peano curve). As the refinement of the curve is a local process, the same method of cell ordering can be applied also to adaptively refined spacepartitioning grids. See the right picture in Fig. 4 for a two-dimensional example. For the parallelisation, it is important that the Peano space-filling curve used for this code is continuous. Thus, the partitions are connected. Furthermore, the resulting partitions exhibit a quasi-minimal surface-volume relation [27, 4].

3 The New Solver Algorithm

3.1 The Basic Algorithm

The new solver algorithm is based on one central algorithmic idea that can be seen as a common building block of all actions of the solver: the cell-wise sweep over the spacepartitioning computational grid in the order prescribed by the Peano curve (see Sect. 2.3). Depending on the current task, different functions such as smoothing, interpolation, restriction, grid refinement or grid coarsening are called for each cell during this sweep. To realise this concept, a cell-wise operator evaluation is required [11]. Thus, we do not have to assemble and store a global system matrix or specialised difference stencils at distinguished nodes[5] The only thing we have to add at such nodes is a correct transport of the cell contributions via restriction and

[5] Nodes with neighbouring cells of different refinement depths.

interpolation to the respective data points with associated degrees of freedom. To explain the principle, we choose the example of the Laplacian evaluated for vertex data. See Fig. 3 for cell-wise evaluation of the five-point stencil. Unlike other approaches as [24], our approach allows for a very easy treatment of hanging nodes. Even scenarios where the parent nodes are hanging nodes themselves are treated without problems.

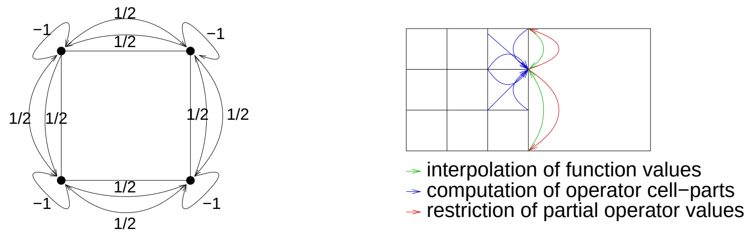
1/2
−1
−1
1/2
1/2
1/2
1/2
1/2
1/2
−1
−1
1/2
1/2
interpolation of function values
computation of operator cell−parts
restriction of partial operator values

Fig. 3 Left: Cell-parts of a scaled five-point stencil approximating the two-dimensional Laplacian; right: interplay between data interpolation, computation of operator cell-parts and restriction of the operator parts from the hanging nodes (no degrees of freedom) to neighbouring degrees of freedom in the adaptive spacepartitioning grid.

The Peano curve defines a cell processing order derived on-the-fly. The locality properties of the Peano curve already lead to a quasi-optimal time locality of data access in the sense that the time between the first and the last access of a datum within one sweep over the grid is short (for the locality properties of self-similar recursively defined space-filling curves and their discrete iterations see [27, 11]). In addition to the time-locality of data access, a good usage of any architecture's cache hierarchy requires also a high spatial locality of data access, i.e. the processing of data structures without 'jumps' in the memory. To achieve this we use stacks, very simple data structures that allow for the two operations `push` a datum on top of the stack and `pop` a datum from the top of a stack. When a vertex is needed for the first time during a traversal, it is read from an input stream holding all data in the order of their first usage. Afterwards, it is `pushed` on another stack holding data temporarily. After the last usage, it is written to an output stream. In the next iteration, we run the Peano curve backwards and, thus, can use the resulting output stream as an input stream.

The idea of the construction of the stacks can be understood best looking at a simple two-dimensional example with a regular grid as displayed in Fig. 4. If we consider the cell vertices at the left-hand or the right-hand side of the Peano curve, we observe that both groups of points are visited in a to-and-fro manner: In one direction during the first pass and in the opposite direction during the second pass of the curve. This corresponds to the stack principle (pile up data and get them back in a last-in-first-out order again). Thus, two stacks are sufficient in this case (one 'right-hand-side' stack and one 'left-hand-side' stack). The same principle can be extended to the three-dimensional case where particular properties of the Peano curve are essential. This is the reason to use this curve and accept the partitioning into three of our grid cells per refinement step and coordinate direction instead of bipartitioning [20]. Furthermore, the data access scheme can be generalised to hierarchical multilevel data and adaptively refined grids with a small number of stacks independent of the refinement depth [10, 20]. Within each stack, the spatial locality of data access is maximal since there is no ’jump’ in memory.

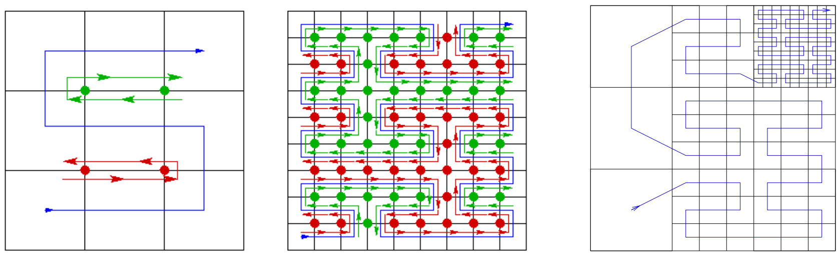

Fig. 4 Left and middle: assignment of the nodal data in a two-dimensional regular Cartesian grid to the two stacks necessary in this case for the intermediate storage during one sweep over the grid. Green points/stack: right-hand side of the curve; red points/stack: left-hand side of the curve; right: two-dimensional adaptive spacepartitioning grids (black) with the associated iteration of the Peano curve (blue).

For the parallelisation of our solver, we use the domain partitioning approach already described in Sect. 2.3. As each of our subdomains is represented by a group of subtrees of the spacepartitioning tree, this requires almost no changes of both the data structure and data access concept.

3.2 Dynamic Adaptivity

The dynamic adaptivity of the computational grid can be integrated without efficiency losses in a very natural way: According to the chosen adaptivity criteria, grid cells are marked for refinement or coarsening during the runtime of the solver. If we access these cells in the next iteration, we interrupt the reading of the input stream to generate new data and directly put them on the appropriate stack. In the case of coarsening, we simply do not write the respective data to the output stream.

3.3 Multigrid

As the Peano curve defines an order of the grid cells on all levels (i.e. the whole cell tree) in a top-down-depth-first manner, methods working on multilevel data can be realised naturally in our concept. Currently, we have two multirid solvers available: An additive and a multiplicative variant. An iteration of the additive multigrid method starts on the coarsest level and interpolates all (hierarchical) values to the finest level (dehierarchisation) during the steps down. Once (locally) arrived on the finest level, we compute the residual in our cell-oriented way and apply the smoother as soon as the computation of the residual is finished. According to the concept of additive multigrid methods, smoothing on all levels is performed simultaneously – without any intermediate updating. Therefore, in the subsequent steps up in the cell tree, we restrict the fine grid residuals to the vertices of the respective father cells. This restriction is performed in a cell-oriented way, too: the vertices of the coarse grid cell collect residual parts from all children of their neighbouring cells. Again, we smooth as soon as the complete restricted residual has been collected. As a result of this additive cycle, we have an updated representation of our unknown function(s) in the respective hierarchical generating system (Fig. 5).

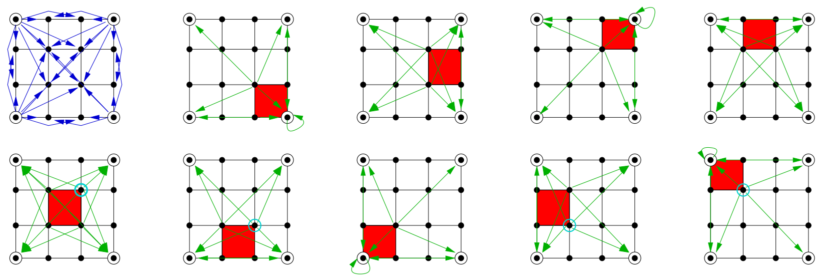

Fig. 5 Schematic visualisation of the computational steps of the additive multigrid method on a coarse grid cell consisting of nine fine grid cells in the two-dimensional case (blue: interpolation; red: computation of the cell-part of the residual in the current fine grid cell; green: restriction, circle around vertices: smoothing) [17].

In contrast to the iterations of the additive multigrid method, an iteration of a complete multiplicative cycle cannot be done within one top-down-depth-first traversal of the grid since we have to finish smoothing on one grid level in all cells neighbouring a vertex before we switch to the next level. Thus, perform a run over the cell tree not going deeper than the current level. To prevent unnecessary processing of fine grid data, we swap out fine grid data to intermediate data structures whenever they are not needed.

3.4 Extensions for a Flow Solver

To build a finite element solver for the incompressible Navier-Stokes equations, we have to integrate some extensions fitting into the general concept: We use a partially staggered grid, that is velocities are associated to the grid's vertices and the pressure is assigned to the midpoints of the grid cells, and a Chorin-like projection method [5]. For the time stepping, we have to solve a pressure poisson equation in each time step. To evaluate the respective Laplacian operator for the pressure in a cell-wise manner, we can apply a two-step scheme: In a first step, we compute the pressure gradients at the cells vertices and in a second step, we use these gradients to finally compute the Laplacian in the cells midpoints [25]. The two steps are merged into one iteration. In a first test-code, we also implemented a multigrid solver for the pressure Poisson equation on the basis of this method [25].

4 The Reference Solver on Regular Grids: F3F

As a stable, well-engineered CFD code for present simulation scenarios and as a reference code to validate new algorithms' results, we developed the fluid dynamics code F3F solving the Navier-Stokes equations based on a finite volume discretisation [8, 3]. F3F consists of modules that can easily be exchanged and works on regular Cartesian grids. Hence, it can use relatively simple data structures and flexible and robust operations.

There are a serial and a parallel F3F version. For the latter, according to the simplicity of the underlying data structures, we decompose the domain into rectangular partitions at the setup phase of the program. This results in a static load-balancing which is sufficient in this case as the program does not allow for dynamic grid adaptivity and, thus, the load per partition does not change.

To solve the pressure Poisson equation in each time step, F3F explicitly assembles the system matrix. Thus, we can integrate and test different solvers for linear systems of equations tuned for various computer architectures (we use PETSc solver, e.g.). As a tailored solver, F3F uses a preconditioned Conjugate Gradient method. This solver is a separate module and, thus, we can apply different parallelisation schemes for the solver and the computational domain. To secure in particular long-run application simulations, checkpointing of the simulation is implemented.

To adapt F3F in a partitioned fluid-structure simulation framework, the program comprises functions to handle and compute forces and displacements at the interface between fluid and structure and provides an interface for FSI simulations in a partitioned approach.

5 Porting F3F to HLRB2. First Performance Results

F3F has been developed having both modularity and portability in mind. Therefore, with respect to portability, it uses development libraries such as Glib or Xerces, commonly available on high-performance computers operated under Linux. The GNU Triangulation Library (GTS) is additionally needed by F3F for manipulating geometry data of the computational domain. The programming paradigm employed for the development of F3F is message passing and, thus, MPI 1 has been chosen for implementing the communication between the parallel parts/elements of the application. To facilitate the software development process of F3F and to reduce the effort of porting the program to different architectures and computing environments, we employ the GNU auto tools suite. Nevertheless, the following issue has been encountered when enabling some of the compiler optimisation techniques available with the Intel Compiler Suite: the Intel Compiler for C/C++ ran out of resources while applying the respective optimisations. This implied a more laborious work in finding appropriate combinations of compiler flags that would lead to a faster F3F on HLRB2.

In Table 3, we show the first parallelisation results obtained on HLRB2 for the drift ratchet scenario (see Sect. 7). They correspond to meshes containing a number of cells ranging form 56,700 to 907,200. The second column contains the time in

# cpu	Time solver (s)	Total time (s)
8	0.32	1.32
16	0.55	2.82
32	1.13	5.89
64	2.77	12.65

Table 3 Scalability of F3F by proportionally increasing the problem size (form 56,700 to 907,200 grid cells) with the number of processors (# cpu). Time split up into time spent in the CG (conjugate gradient) solver and total time.

seconds needed for the computation of the pressure field in each time step of the flow simulation by the solver for the linear system of equations available in F3F. The third column holds the total time needed by our flow program for one complete time step. It includes also the time values for the solver. Simulations for other scenarios exhibit a similar scaling.

For computing flow simulations with the current version of F3F and the tailored CG, the use of a number of processors larger than in Table 3 should be carefully considered. Despite an efficient implementation of the solver, CG's implicit synchronisation points cannot be eliminated and, thus, the scalability of the algorithm remains rather limited.

6 Flow Simulation on Adaptive Grids

A finite element approach for the solution of the incompressible Navier-Stokes equations has been implemented for two- and three-dimensional, adaptive Cartesian grids within the new Peano solver package as described in Sect. 3. The method of consistent forces (see [3, 9]) is implemented to get both computationally cheap and accurate force data on the surface of objects (i.e. on the coupling surface in the case of FSI simulations). In all simulations performed, we achieved cache-hit rates above 98% or, seen from the other side, only 110% of the unavoidable number of cache misses given by the requirement to load all data to the cache at least once [10, 20, 11, 19, 25, 3] and managed to do with very low storage requirements (compare Table 1).

Several benchmark computations have been performed in order to check the correctness of the adaptive flow simulations. As an example, the resulting velocity field for the well-known DFG cylinder benchmark 2D-1 [23] at Reynolds number $Re = 20$ for two dimensions with an a priori refined grid is shown in Fig. 6.

Our spacetree approach allows for an isotropic grid refinement at the walls of a channel flow, e.g., as it might be used in turbulent scenarios. A complete spatial

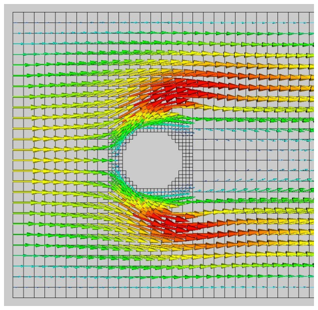

Fig. 6 Visualisation of an adaptive fluid simulation result: Zoom of a 2D adaptive spacetree grid with velocities of the DFG cylinder benchmark scenario 2D-1 [23].

resolution of the regions near the walls is one way to enable more detailed boundary layer investigations. Fig. 7(a) shows a similar test setup in two dimensions where the refinement not only towards the wall but also in flow direction is clearly visible. For this scenario, four grid levels have been used resulting in 17577 fluid cells, 13284 vertices holding degrees of freedom, and 8424 hanging nodes. The corresponding streamlines and pressure distribution are shown in Fig. 7(b). Of course, we observe the typical laminar velocity profile as this setup is still two-dimensional and the Reynolds number is lower than the critical one for the turbulent channel flow scenario. Real turbulence will appear in tree-dimensional direct numerical simulations that currently are in preparation.

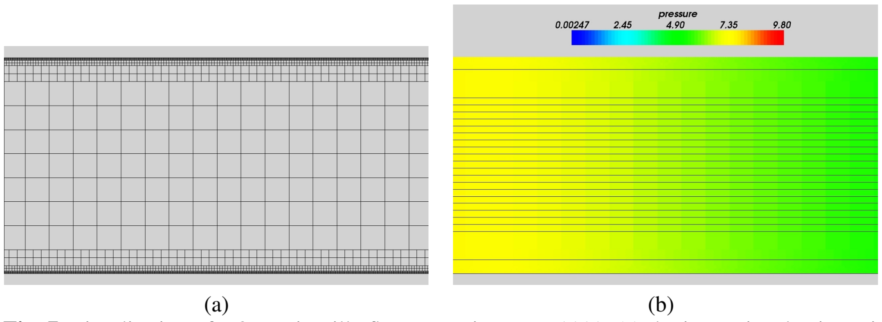

Fig. 7 Visualisation of a 2D Poiseuille flow scenario at Re=1111: (a) the isotropic adaptive grid refinement in all directions near the channel walls and (b) the corresponding streamlines over the pressure distribution.

7 Fluid-Structure Interactions: The Drift Ratchet Scenario

The directed transport of micro-particles depending on their size is the basis for particle sorting methods that are of utmost importance for example in life sciences. A drift ratchet is a so-called Brownian motor that allows for such a directed transport. Hereby, the particle motion is induced by a combination of the Brownian motion and asymmetries stemming for example from the domain's geometry, electrical fields, or transient pressure boundary conditions. We simulate a particular drift ratchet which consists of a matrix of pores with asymmetrically oscillating diameter wherein a fluid with suspended particles is pumped forward and backward and where the particles' long-term transport direction depends on their size. Thus, this setup allows for a continuous and parallel particle separation, which has already been shown experimentally [16]. However, for a deeper understanding and for an optimised parameters' choice, further investigations, i.e. simulations, are necessary.

The drift ratchet simulations turn out to be computationally very expensive and, thus, require both an efficient simulation code and supercomputers. The computational costs stem from the nature of the simulated scenario (large simulation times with small time steps, multi-scale models and multiphysics phenomena, moving and complex geometry)[2].

The scenario we used here consists of a pore geometry with two chambers similar to the one displayed in Fig. 8. The fluid parameters used in our scaled simulation scenario are: density $\rho = 10^3\,\mathrm{kg/m}^3$, dynamic viscosity $\eta = 10^{-3}\,\mathrm{Pa\,s}$. Additionally, we set the maximal mean velocity imposed by the pressure at the inflow to $u_{\mathrm{mean}} = 0.1\mathrm{m/s}$, the characteristic length (minimum diameter of the pore) and the particle diameter to $1\,\mu\mathrm{m}$ and $0.6\,\mu\mathrm{m}$, ending up with $Re = 0.1$.

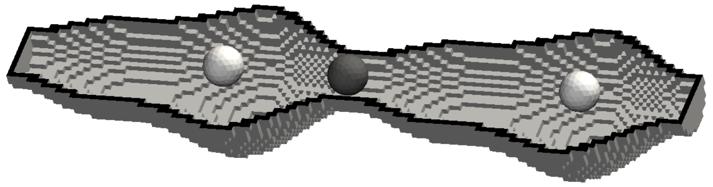

Fig. 8 Three-dimensional asymmetric pore with two chambers. The particle is displayed at three different time steps (i.e. different locations) and the color of the particle corresponds to its velocity (light color – low, dark color – high).

We simulated two scenarios both with oscillating pressure boundary conditions. On the left-hand side of the pore we set the pressure to be zero and on the right-hand side the pressure oscillated within the range of $p_{\mathrm{min}} = -11\,\mathrm{kPa}$ and $p_{\mathrm{max}} = 11\,\mathrm{kPa}$. In the first scenario (Fig. 9, left), the frequency f was set to 10kHz and the particle was placed in the narrowed region between the two chambers. In the second scenario (Fig. 9, right), the frequency f was set to 14kHz with the particle initially positioned in the first chamber.

The drift of the particle from the first chamber to the second one occurs only in the first setup. In the second one, the particle remains in the first chamber. This shows that our numerical approach works for drift ratchet simulations. For understanding the physics behind this sort of experiments, we plan to proceed with parameter studies for our simulations. These will focus on the oscillation frequency of the boundary, on the particle size, on the resolution of the computational mesh, and on the number of pore chambers. The computing resources required for the study of the drift ratchet simulations can be provided only by supercomputers such as HLRB2.

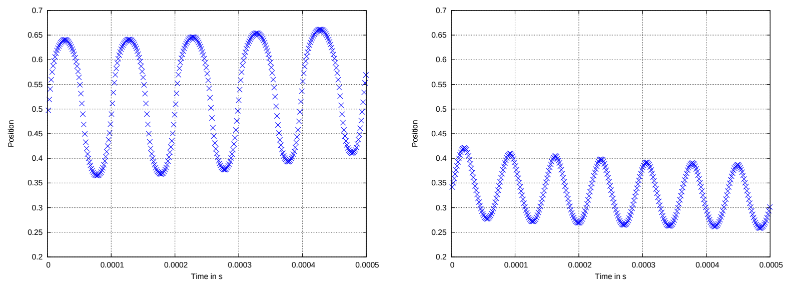

Fig. 9 Relative position of a particle in an asymmetric pore with two chambers, using an oscillating boundary condition $f = 10$ kHz (left) and $f = 14$ kHz (right), respectively. The position is relative to the length of the pore.

8 Conclusion

From our experience with PDE solvers on adaptive Cartesian grids, we conclude that there is a high potential for numerically and hardware efficient solver implementations on these grids. In terms of memory efficiency, our codes already come along with low memory requirements and a high cache-hit rate. The results achieved show the general applicability of Cartesian grids for fluid-structure interactions, i.e. applications that require a high accuracy at the domain boundaries for the computation of forces acting on structures, e.g. Right now, we implement the three-dimensional and parallel Navier-Stokes solver on adaptively refined grids.

Afterwards, we have to improve the runtime of our code by addressing also other efficiency aspects besides pure memory access and consumption. The task will be to run large simulations for various applications ranging from direct numerical simulations of turbulent channel flow to fluid-structure interaction scenarios with a good scalability on the HLRB2. In addition, we make our flow solvers (F3F and Peano) run in the DEISA grid computing environment, with the aim to perform distributed partitioned fluid-structure interaction simulations.

References

1. Timothy J. Barth. Computational fluid dynamics, structural analysis and mesh partitioning techniques - introduction. In *VECPAR '98: Selected Papers and Invited Talks from the Third International Conference on Vector and Parallel Processing*, pages 171–175, London, UK, 1999. Springer-Verlag.
2. M. Brenk, H.-J. Bungartz, M. Mehl, I. L. Muntean, T. Neckel, and T. Weinzierl. Numerical simulation of particle transport in a drift ratchet. *SIAM J. Sci. Comp.* , 2007. in review.
3. M. Brenk, H.-J. Bungartz, M. Mehl, and T. Neckel. Fluid-structure interaction on cartesian grids: Flow simulation and coupling environment. In H.-J. Bungartz and M. Schäfer, editors, *Fluid-Structure Interaction*, number 53 in LNCSE, pages 233–269. Springer, 2006.
4. H.-J. Bungartz, M. Mehl, and T. Weinzierl. A parallel adaptive Cartesian PDE solver using space–filling curves. In E. W. Nagel, V. W. Walter, and W. Lehner, editors, *Euro-Par 2006,*

Parallel Processing, 12th International Euro-Par Conference, volume 4128 of *LNCS*, pages 1064–1074, Berlin Heidelberg, 2006. Springer-Verlag.

5. A. J. Chorin. Numerical solution of the Navier-Stokes equations. *Math. Comp.*, 22:745–762, 1968.
6. K. Daubner. Geometrische Modellierung mittels Oktalbäumen und Visualisierung von Simulationsdaten aus der Strömungsmechanik. Studienarbeit, Universität Stuttgart, Universität Stuttgart, 2005.
7. C. C. Douglas, J. Hu, M. Kowarschik, U. Rüde, and C. Weiss. Cache optimization for structured and unstructured grid multigrid. *Electron. T. Numer. Ana.*, 10:21–40, 2000.
8. M. Emans and Ch. Zenger. An efficient method for the prediction of the motion of individual bubbles. *Int. J. Comp. Fluid Dyn.*, 19:347–356, 2005.
9. P. M. Gresho and R. L. Sani. *Incompressible Flow and the Finite Element Method*. John Wiley & Sons, 1998.
10. F. Günther. *Eine cache-optimale Implementierung der Finiten-Elemente-Methode*. PhD thesis, Institut für Informatik, TU München, 2004.
11. F. Günther, M. Mehl, M. Pögl, and C. Zenger. A cache-aware algorithm for PDEs on hierarchical data structures based on space-filling curves. *SIAM J. Sci. Comput.*, 28(5):1634–1650, 2006.
12. D. Hackenberg, R. Schöne, W. E. Nagel, and S. Pflüger. Optimizing OpenMP parallelized DGEMM calls on SGI Altix 3700. In E. W. Nagel, V. W. Walter, and W. Lehner, editors, *Euro-Par 2006, Parallel Processing, 12th International Euro-Par Conference*, volume 4128 of *LNCS*, pages 145–154, Berlin Heidelberg, 2006. Springer-Verlag.
13. F. H. Harlow and J. E. Welch. Numerical calculation of time-dependent viscous incompressible flow of fluid with a free surface. *Physics of fluids*, 8(12):2182–2189, 1965.
14. W. Herder. Lastverteilung und parallelisierte Erzeugung von Eingabedaten für ein paralleles cache-optimales Finite-Element-Verfahren. Diploma thesis, Institut für Informatik, TU München, 2005.
15. Donald E. Knuth. The genesis of attribute grammars. In *WAGA: Proceedings of the international conference on Attribute grammars and their applications*, pages 1–12, New York, NY, USA, 1990. Springer-Verlag New York, Inc.
16. S. Matthias and F. Müller. Asymmetric pores in a silicon membrane acting as massively parallel brownian ratchets. *letters to nature*, 424:53–57, 2003.
17. M. Mehl, T. Weinzierl, and C. Zenger. A cache-oblivious self-adaptive full multigrid method. *Numer. Linear Algebr.*, 13(2-3):275–291, 2006.
18. R.-P. Mundani. *Hierarchische Geometriemodelle zur Einbettung verteilter Simulationsaufgaben*. Berichte aus der Informatik. Shaker Verlag, Aachen, 2006. PhD thesis.
19. T. Neckel. Einfache 2d-Fluid-Struktur-Wechselwirkungen mit einer cache-optimalen Finite-Element-Methode. Diploma thesis, Fakultät für Mathematik, TU München, 2005.
20. M. Pögl. *Entwicklung eines cache-optimalen 3D Finite-Element-Verfahrens für große Probleme*, volume 745 of *Fortschritt-Berichte VDI, Informatik Kommunikation 10*. VDI Verlag, Düsseldorf, 2004.
21. H. Sagan. *Space-filling curves*. Springer, New York, 1994.
22. M. F. Tomé and S. McKee. GENSMAC: A computational marker and cell method for free surface flows in general domains. *J. Comp. Phys.*, 110:171–186, 1994.
23. S. Turek and M. Schäfer. Benchmark computations of laminar flow around a cylinder. In E. H. Hirschel, editor, *Flow Simulation with High-Performance Computers II*, number 52 in NNFM. Vieweg, 1996.
24. W. Wang. Special bilinear quadrilateral elements for locally refined finite element grids. *SIAM J. Sci. Comput.*, 22(6):2029–2050, 2001.
25. T. Weinzierl. Eine cache-optimale Implementierung eines Navier-Stokes Lösers unter besonderer Berücksichtigung physikalischer Erhaltungssätze. Diploma thesis, Institut für Informatik, TU München, 2005.
26. G. Wellein, T. Zeiser, and P. Lammers. Application performance of modern number crunchers. *CSAR Focus*, 12:17–19, 2004.
27. G. Zumbusch. Adaptive parallel multilevel methods for partial differential equations. Habilitationsschrift, Universität Bonn, 2001.